IRRES: Intrusion-Resilient Remote Email Storage

Di Ma
Computer and Information Science Department
University of Michigan-Dearborn
Email: dmadma@umich.edu

Gene Tsudik
Computer Science Department
University of California, Irvine
Email: gts@ics.uci.edu

***Abstract*—Both individuals and corporations increasingly rely on email to exchange important and, often sensitive, information. With the advent of ubiquitous computing and miniaturization of end-devices, many users leave email on remote servers, thus facilitating anywhere/anytime access from any networked device. Since private and sensitive information is often contained in email, it is very important to prevent its unauthorized disclosure.**

In this paper, we consider the problem of repeated intrusions of third-party email servers. We present IRRES: Intrusion-Resilient Remote Email Storage system that prevents email leakage in the presence of a powerful mobile adversary. IRRES eliminates sender compliance requirements by shifting encryption from email sender to email server. Incoming email messages are encrypted by the server with one-time keys obtained from an intrusion-resilient cooperative randomness generation process. Thus, even if the server is compromised twice, all email encrypted in the interim remains secure.

***Keywords*-intrusion resilience, PRNG, mobile adversary, cloud computing, remote email storage**

I. Introduction

Constantly increasing amounts of data (such as email, transaction records, and documents) are being outsourced into the proverbial "cloud" due to the increasing popularity of cloud computing. Benefits of this trend include: low entry barriers, shared infrastructure yielding lower costs, lower management overhead, constant anytime/anywhere access, as well as better reliability.

In fact, even the federal government is interested in using cloud computing as a way to increase "innovation, efficiency, and effectiveness in federal IT" [1]. In the foreseeable future, cloud storage will host a huge amount of data from both public and private sectors. The online nature of cloud storage, the sensitive information of the stored data and the complicated engineering process of cloud platform implementation engender a wide range of both internal and external threats. Although cloud storage is believed to be more secure and reliable than traditional "in-house" storage, examples of outages and security breaches of cloud services start to appear [2]–[4].

A successful attack, whether originating inside or outside the cloud provider, causes information leakage which, in turn, can have very serious consequences. They might include: privacy violation, identity theft, legal liability, loss of customers and lowered employee productivity. It is thus imperative for sensitive information to be well-protected, for the sake of both cloud users and owners/operators. Since storage is the cloud computing's most basic and natural application, its security is crucial to the future of cloud computing.

We distinguish among two types of cloud-stored data: *push* and *pull*. The former represents outsourced data, i.e., data originated by the user and later outsourced to the cloud. It typically represents archival or backup data, such as the well-known "Storage-as-a-Service" concept. Whereas, *pull* data is generated within the cloud and later retrieved by the user. The research community has, thus far, focused on protecting *push* data, considering authentication, integrity, confidentiality and retrievability to defend against untrusted cloud storage servers [5]–[12]. Much less attention has been paid to security of *pull* data. As more and more *pull* data is generated through various "XXX-as-a-Service" applications [1], it is important to protect this data from the very beginning, i.e., from the time of its "*birth*" in the cloud.

Unlike *push* data which first resides at the user side (thus, its security can be enforced before outsourcing), user-imposed security is difficult to enforce for *pull* data, since it exists before the user sees it. We claim that differences between *push* and *pull* data motivate substantially different security approaches.

A. Motivation and Focus

In this paper, we focus on one specific type of *pull* data - email. Email is one of the most popular Internet applications. It increasingly represents official communication; it is also often subpoenaed and treated as legal evidence. Suffice it to say that email messages have become a form of valuable and sensitive private property for both individuals and larger (e.g., corporate and government) entities. Advances in pervasive and ubiquitous computing have yielded smaller and smarter end-devices, such as cell-phones and PDAs. Because many users armed with such (limited-storage) devices are also highly mobile, they often choose to leave email on remote servers, thus benefiting from anywhere/anytime access from their networked end-devices. Since sensitive information is often contained in email, there is a natural cause for concern that personal or proprietary information might be leaked due malicious storage server access.

Of course, outsourced email can be secured by using end-to-end security schemes, such as PGP or S/MIME. However,

[1] where "XXX" denotes any application, such as: Email, transaction management, sensing, RSS feeds and publish-subscribe

such schemes require explicit and universal sender compliance. In other words, each sender has to know public keys of **all** entities it communicates with. To send an email, the sender needs to encrypt it with the receiver's public key. For multi-destination email, the sender has to encrypt the message (or at least a message-specific symmetric key) for every recipient. Furthermore, though it is generated by the sender, once an email is transmitted, it is considered to be property of the receiver(s). The receiver is the entity concerned with the security of its email stored on the server. Inability to enforce universal compliance and scalability issues on the sender side have resulted in marginal acceptance of public key-based end-to-end email encryption.

This paper considers security of outsourced (cloud-resident) email in the presence of powerful re-visiting adversaries which repeatedly re-visit once-compromised cloud servers to harvest new data. To this end, we develop IRRES – Intrusion Resilient Remote Email Storage, a system that secures incoming emails once they arrive at the cloud email server. We note that many data breach laws (in various states) require notification be sent to customers whenever a data breach occurs, except if the lost or leaked data is encrypted [13]. Therefore, it is also in the cloud provider's interest to encrypt each incoming email entering its cloud. Key IRRES features:

- IRRES obviates the need for active and universal sender compliance. Senders transmit email as usual, while the destination email server deals with encryption. Regardless of the sender, email content is protected once it reaches the server.
- IRRES is intrusion-resilient. In IRRES, email is encrypted by the server with keys generated such that they remain secret even in the face of a powerful re-visiting (mobile) adversary.
- IRRES does not rely on trust models, such as a global PKI used by S/MIME or web-of-trust technology used in PGP [14].

To achieve intrusion-resilience, we exploit network connectivity and the cooperative environment within the cloud infrastructure. The email server re-initializes its secure state (which is used to generate encryption keys) periodically by cooperating with its peers in the infrastructure. Once its secure state is refreshed, a returning adversary, knowing the server's secure state from some previous time, cannot compute any subsequent secrets used by the server. This implies intrusion-resilience.

The rest of this paper is organized is as follows. Section II presents our trust and adversary models. Section III describes the design of IRRES. Security analysis and performance analysis are in Sections IV and V, respectively. Section VI reviews related work and Section VII concludes the paper.

II. Trust and Adversary Models

In this section, we present our trust and adversary models.

A. Trust Model

Our trust model resembles the one used in current telephone networks where a network operator is considered trusted. Like a telephone network operator (e.g., AT&T or Verizon), a cloud provider (e.g., Amazon or Google) is usually a large commercial entity that cares (one hopes) about its reputation and brand name value. For the same reason, a cloud provider is trusted to provide appropriate service, as specified in the business contract between it and the user.

We consider a set of peer email *s*ervers connected to each other over the Internet but managed by a common administration – the cloud *p*rovider. Based upon trust in the provider, an email server is trusted to faithfully receive, store and serve email to its users. However, it is not immune to compromise. At any given time, each server holds email for users (subscribers) who register their public keys with the server. A server can be accessed remotely by its users or locally (e.g., by administrators). It is thus subject to both outsider and insider attacks. Our goal is to prevent data leakage due to both.

B. Adversary Model

We envision with a re-visiting (mobile) adversary that aims to learn as much data as possible. It might gain access to an email server through different channels. An outsider can do so by exploiting system vulnerabilities. Real-world experience shows that, no matter how well a system is protected, there is always a way for a sufficiently motivated (and funded) adversary to find a back door. An insider may already have legitimate right to access the server. Insiders are more dangerous because they have more information about the system. Consequently, it is more difficult to mitigate insiders. Regardless of its origin, a successful attack causes information leakage. Once an adversary compromises a server, it reads all local storage and memory (including any and all secrets), and listens to all its communication channels.

Email data is usually very dynamic in nature. It is thus in the adversary's interest to re-visit once-compromised server to harvest new data. A re-visiting adversary is motivated to minimize destructive actions, such as changing server code or installing malware. This is in order to stay stealthy (undetected) since doing so allows it to re-visit the server again and again. A re-visiting adversary does not stay on the server for very long, since lingering increases its chance of being detected. Instead, it releases the server for a limited time, re-compromises it later and expects to learn all email data that has arrived in the interim..

III. IRRES: Main Idea

The first step to reduce the impact of email server compromise is to minimize the number of secrets kept on the server. This naturally suggests the use of public key encryption. That is, the server encrypts every incoming email using the recipient's (email cloud user's) public key. Because the server does not keep any private keys, the adversary who gains

control of the server can not learn any user's private keys and thus cannot decrypt encrypted email content directly.

However, as is well-known, public key encryption is rarely used on bulk data. Instead, hybrid (or envelope) encryption is used: the email message is encrypted using some symmetric scheme under a unique secret key and the latter is itself encrypted using the recipient's public key. This hybrid encryption approach is common practice, used in many systems, such as PGP and S/MIME. Per message symmetric keys are generated randomly, such that knowing one key does not lead to learning another.

However, there is a critical issue that has to do with how such message-specific symmetric keys are generated. If they are obtained from a True Random Number Generator (TRNG), it is easy to see that a re-visiting adversary can not learn any keys used during its absence. This is because a TRNG generates information-theoretically independent values based on some physical phenomena.

Informally, given an arbitrarily long sequence of consecutive TRNG-generated values, removing any one number from the sequence makes any guess of the missing value equally likely. In other words, learning a value in a given position of the TRNG-generated sequence, provides no information on values in either previous or following positions. Let K^r be the symmetric key used in round r. Suppose that an adversary compromises the server at round r' and leaves it at round $r'' > r'$. Encrypted data from any round $r < r'$ remains secret, since the data is encrypted under K^r and all keys – obtained from a TRNG – that the adversary learns while in control of the server are statistically independent from $K^{r'}$. Thus, we have **forward secrecy** – secrecy of data generated before compromise. Similarly, any data encrypted after round r'' (after the adversary leaves the server) also remains secret, since all random numbers the adversary learns while in control of the server are statistically independent from those generated later. Thus, we have **backward secrecy**. To summarize, if the server is equipped with a TRNG, intrusion-resilience can be trivially achieved.

On the other hand, if symmetric keys are obtained from a pseudo-random number generator (PRNG), the resulting security is dubious. This is because a typical PRNG produces "pseudo-random" numbers by starting with a (secret) seed and using some function to produce a sequence of values that *appear* random. Often, one-way functions (functions that are computationally infeasible to invert) are used as PRNGs. Thus, a typical PRNG provides forward security. However, given access to the current state of the PRNG, it is easy to compute all subsequent values. Thus, even after the adversary leaves the compromised server, the server's state is easily reconstructible. Consequently, there is no backward security.

True random number generation requires one or more physical sources of randomness. Thermal noise, keyboard typing patterns, biometrics have been used for this purpose [15]. However, in many cases, such a physical source may not be readily available or be sufficiently good for true random number generation. Therefore, we only assume existence of a PRNG on the server.

Given the above assumptions, our challenge is how to let the server generate strongest possible values using a PRNG. Our approach involves settling for sources of "fresh" pseudo-randomness, instead of "true" physical randomness. By "fresh", we mean *unknown to the adversary*. The server updates its PRNG state with this fresh randomness and, as the latter is unknown to the adversary, it can not compute keys used after the server's state is refreshed. To obtain fresh randomness, we exploit network connectivity and the cooperative environment where peer servers are willing to assist one another to refresh PRNG state.

A. Cooperative State Refresh Protocol

We assume that each server can communicate with another $n-1$ peer servers. The entire n servers form a co-op. Co-op members do not have any common secrets (i.e., they share no keys). However, they are required to maintain loosely synchronized clocks. All members run the same refresh protocol.

All nodes in the co-op are equipped with a PRNG initialized with a distinct secret seed. Although random numbers generated from the same PRNG are internally dependent, random numbers generated from different PRNGs are mutually independent. So, a random number generated by a peer can be used as a source of fresh randomness to re-initialize the PRNG of other nodes.

Since we do not know the exact time when a particular node is compromised, the cooperative state refresh protocol is conducted periodically. Time is divided into equal size epochs. At the end of each epoch, a node refreshes its PRNG state through the cooperative state refresh protocol before it moves to the next epoch. With the infusion of fresh randomness from its peers, a node generates independent random numbers in different epochs.

The membership of a co-op should be based on achieving maximum overall diversity. The more diverse (located in different geographical locations, running on different platforms with different operating systems, etc.) the co-op, the better security it attains.

The cooperative state refresh protocol consists of two sub-algorithms: *contribute* and *refresh*, as shown in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1: Node i CONTRIBUTE

1 Generate contributions: $\{r_{i,1}, \cdots, r_{i,n}\} \leftarrow PRNG_i$
2 Keep $r_{i,i}$
3 Send contribution $r_{i,j}$ to node j ($j = [1, n]$ AND $j \neq i$)

Algorithm 2: Node i REFRESH

1 Receive $r_{j,i}$ from j ($j = [1, n]$ AND $j \neq i$)
2 Calculate $r = \oplus_{j=1}^{n} r_{j,i}$
3 Initialize $PRNG_i$ with r

Every node runs these two algorithms in the end of each epoch. Frequency of state refresh is based on the perceived

level of risk. For example, it can be based on the number of email messages processed or on time elapsed. In *contribute*, a node i generates a random contribution for every node in the co-op, including itself, based on its current PRNG state. It keeps $r_{i,i}$ for itself for later use, and sends $r_{i,j}$ to peer j. In *refresh*, node i receives random numbers generated by its peers and uses them along with a self-contribution $r_{i,i}$ to compute an aggregate r. It then initializes its PRNG with this newly generated value. The whole process is cooperative: a node helps others in *contribute* and obtains help in *refresh*.

We note the inclusion of self-contribution $r_{i,i}$ in node i's PRNG initialization is very important since it guarantees that node i's secure state will not be determined by the infusion of tainted contributions - those that are known to the adversary. In IRRES, only fresh contributions help a node recover from being compromised, while tainted contributions have no effect on a node's secure state.

B. Multi-Destination Email

When a multi-destination email arrives to the server, it has two ways to process it, with different security implication. In the first approach, the server encrypts the email once and the same encrypted copy of the email is shared among all recipients. In the second approach, the server generates a different symmetric key and uses it to produce one encrypted copy for each receipt. The second approach provides slightly better security, since the adversary can not distinguish between a multi-destination email and an individual email. Whereas, the first approach leaks information about the group of recipients.

IV. Security Considerations

Suppose that the adversary can compromise k co-op nodes within each epoch. To learn as many secrets as possible, the best strategy for the adversary is to always compromise a new set of nodes in the beginning of each epoch (after observing the cooperative state refresh process of the controlled node). This way, the adversary learns secrets of $2k$ nodes: k nodes compromised in the previous epoch and k nodes currently under its control, before the next state refresh process finishes. Assuming this adversarial mobility strategy, we present the following two facts that characterize intrusion-resilience of of IRRES.

> FACT 1: The cooperative state refresh process does not help nodes recover secrecy after two epochs, if the adversary can compromise more than $\lfloor \frac{n}{2} \rfloor$ at each epoch.

This reflects the situation whenever the cooperative state refresh process does not help nodes recover from compromise. If $k > \lfloor \frac{n}{2} \rfloor$, after two rounds, the adversary learns all secrets of all nodes in the co-op and no fresh randomness remains in the system. Since tainted randomness does not help, the cooperative state refresh process fails.

> FACT 2: If the adversary can compromise less than $\lceil \frac{n}{2} \rceil$ nodes in the co-op at each epoch, a node, compromised and released, can recover from being compromised after the next state refreshing process.

If $k < \lceil \frac{n}{2} \rceil$, there is at least one node at each epoch whose state is unknown to the adversary. This node can serve as a source of fresh randomness and help a set of other nodes (previously compromised) recover their secure state. Once this happens, these nodes' PRNGs are freshly initialized.

V. Performance Analysis

The cooperative state refresh process consists of only computation efficient operations such as hashes and exclusive-ORs, assuming we use a cryptographic hash function (e.g., SHA-2) as a PRNG. A node performs n hash operations to generate n contributions. Since every node communicates with all others in its co-op, a total of $(n-1)^2$ messages are needed per refresh process. In practice, we do not expect n to be large, since k is expected to be small - considering the difficulty to compromise multiple nodes in different geographical locations and with different operating systems.

We can trade off some security to achieve better communication overhead. Note that only a single fresh contribution is needed to "cure" a node. Once a fresh contribution is received, further contributions do not provide any additional help. Thus, instead of generating contributions for all $n-1$ peers, a node can randomly pick $t << n$ peers to help in *contribute*. Since each node would pick t peers randomly, if t is small, there is a danger of having some nodes that receive no contributions at all. To ensure that each node gets at least one contribution, we need to tune t to to achieve a balance between security and performance. Based on the well-known Coupon Collector's problem, we can set $t = ln(n) + c$, so that the probability a node receives no contribution is e^{-c}. This way, we can reduce communication overhead from $(n-1)^2$ to nt while maintaining security comparable to that of the general approach.

VI. Related Work

Previous work most related to ours is the SSARES (Secure Searchable Automated Remote Email Storage) scheme proposed in [16]. SSARES also encrypts incoming email on the server side. However, it differs from IRRES in several ways. While SSARES uses PEKS (Public-key Encryption with Keyword Search) to support keyword search, it uses RSA for email encryption. Consequently, a user needs to register two sets of public keys with the server. Furthermore, SSARES does not provide protection against re-visiting (mobile) adversaries. A re-visiting adversary that learns the PRNG state of the server at some time, can compute all randomness generated since that time. Knowledge of this randomness allows the adversary to verify its guesses of plaintext encrypted under RSA and PEKS. This is particularly the case if the message space of PEKS encryption – the keyword space – is small.

Recovering from being compromised under the presence of mobile adversary has been studied in [17]. In this proactive scheme, each node communicates with all of its peers to update its secure state at the end of each round, just like the state refresh process in IRRES. However, [17] assumes the *disappearance* of the adversary between the time the adversary

releases compromised nodes and the time it moves to a new set of nodes. If the adversary can not compromise all n nodes, there is always at least one that can help others recover. With the absence of the adversary during that period, all nodes can recover from compromise and the maximum number of nodes with states known to the adversary is exactly k at all times. Therefore, this scheme is $n-1$ resilient. Our approach does not assume that the adversary is absent at any time; indeed, we consider it to be unrealistic. Instead, we assume that even when the adversary releases a node, it still remembers its secret state. This slight difference between adversary models makes our analysis and evaluations quite different.

Distributed key update is also studied in the context of unattended wireless sensor networks [18], [19]. In [18], sensors explicitly ask peers for help, while in [19], sensors volunteer help to others. Due to the resource constrained property of sensors, these schemes only provide probabilistic security since sensors only collaborate with a small subset of sensors to reduce communication overhead incurred by collaboration. In wireless communication, eavesdropping is easy. So the authors analyze the effect of eavesdropping on cooperative key update.

Our work is also related to outsourced data security. One body of research allows a client that has stored data at a untrusted server to verify that the server possesses its original data without retrieving the data. This is achieved by the server sending back a Proof of Possession (PoP) to the client each time when the client queries. The topic is considered in both the public and conventional key contexts [7]–[12]. Another body of research concerns security (integrity, authenticity and completeness) of query results answered by outsourced data servers [5], [6].

VII. Conclusion

In this paper, we propose Intrusion Resilient Remote Email Storage (IRRES), a system that remains secure under the presence of system compromise and automatically recovers from being compromise by letting the server engage in a cooperative state refresh process with other peers in the infrastructure. Our future work includes system implementation and optimization of server side operations.

References

[1] "Crosscutting programs," http://www.whitehouse.gov/omb/budget/fy2010/assets/crosscutting.pdf.

[2] Amazon.com, "Amazon s3 availability event: July 20, 2008," Online at http://status.aws.amazon.com/s3-20080720.html, July 2008.

[3] Scott Wilson, "Appengine outage," Online at http://www.cio-weblog.com/50226711/appengine_outage.php, June 2008.

[4] Brian Krebs, "Payment processor breach may be largest ever," Online at http://voices.washingtonpost.com/securityfix/2009/01/payment_processor_breach_may_b.html, Jan. 2009.

[5] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," in *NDSS 2004*, Feb. 2004.

[6] HweeHwa Pang, Arpit Jain, Krithi Ramamritham, and Kian-Lee Tan, "Verifying completeness of relational query results in data publishing," in *SIGMOD'04*, 2005, pp. 407–418.

[7] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song, "Provable data possession at untrusted stores," in *ACM CCS'08*, 2008.

[8] G. Ateniese, R. Di Pietro, L. Mancini, and G. Tsudik, "Renewable traitor tracing: a trace-revoke-trace system for anonymous attack," in *Securecom'08*, 2008.

[9] Q. Wang, C. Wang, J. Li, K. Ren, and W. Lou, "Enabling public verifiability and data dynamics for storage security in cloud," in *ESORICS'09*, 2009.

[10] Chris Erway, Charalampos Papamanthou, Alptekin Kupcu, and Roberto Tamassia, "Dynamic provable data possession," in *ACM CCS*, 2009.

[11] Kevin Bowers, Ari Juels, and Alina Opre, "Hail: A high-availability and integrity layer for cloud storage," in *ACM CCS*, 2009.

[12] Cong Wang, Qian Wang, Kui Ren, and Wenjing Lou, "Privacy-preserving public auditing for data storage security in cloud computing," in *INFOCOM'10*, 2010.

[13] S. Berinato, "Data breach notification laws, state by state," Online at http://www.csoonline.com/article/print/221322, Jul. 2008.

[14] S. Garfinkel, *PGP: Pretty Good Privacy*, O'Reilly and Associate, 12 1991.

[15] Jon Callas, "Using and creating cryptographic-quality random numbers," Online at http://www.merrymeet.com/jon/usingrandom.html, June 1996.

[16] Adam Aviv, Michael Locasto, Shaya Potter, and Angelos Keromytis, "Ssares: secure searchable automated remote email storage," in *The 23rd Annual Computer Security Applications Conference (ACSAC)*, 2007, pp. 129–138.

[17] R. Canetti and A. Herzberg, "Maintaining security in the presence of transient faults," in *14th Annual IACR Crypto Conference (Crypto'94)*, 1994, pp. 425–438.

[18] D. Ma and G. Tsudik, "DISH: Distributed self-healing in unattended wireless sensor networks," in *10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'08)*, 2008, pp. 47–62.

[19] R. Di Pietro, D. Ma, C. Soriente, and G. Tsudik, "POSH: Proactive co-operative self-healing in unattended sensor networks," in *27th IEEE International Symposium on Reliable Distributed Systems (SRDS'08)*, 2008, pp. 185–194.